UNIVERSITÉ MONTPELLIER 2
SCIENCES ET TECHNIQUES

Master Informatique
Année universitaire 2007-2008

Rapport de projet

Plugin Madkit pour Eclipse





Encadrants :
M. Jacques Ferber – M. Tiberiu Stratulat

Groupe n°10 :
Matthieu Carrat – Matthieu Gaufrès – Guilhem Molla – Dimitri Prikhodko

SOMMAIRE

Introduction

Ce document va essayer de présenter une bonne partie de notre travail réalisé au cours de ce TER.

Le projet consistait en la réalisation d'un plugin sous Eclipse permettant d'aider à la réalisation de programmes orientés agents sous Eclipse, ces agents fonctionnant sur la plate-forme MadKit.

Le rapport se décompose en plusieurs parties : les principales traitant de nos débuts (la mise en marche avec la découverte des plugins existants, adaptation, …), des points importants de notre apprentissage nécessaires à la réalisation du projet, les nouvelles fonctionnalités que nous avons apportées au plugin, un bilan critique (objectifs atteints, améliorations possibles, organisation …). Il se termine par un glossaire, des références et remerciements.

Tout au long de notre rédaction, nous essayerons de faire part des problèmes auxquels nous avons été confrontés et les solutions que nous avons envisagées ou même mis en place.

1 – Analyse de l'existant

1.1 – Mise en marche

Nous nous sommes séparés en binômes pour étudier les deux plugins déjà créés précédemment par des étudiants (MadPlugin par Quentin Caron et Madkit Plugin par Christian Plutis).

Nous nous sommes rapidement rendu compte que le plugin de Christian Plutis était plus complet et nous offrait une base pour démarrer plus intéressante. De plus, nous possédions les sources afin de nous faire une idée de sa structure.

La reprise du plugin a nécessité quelques manipulations pour le faire fonctionner, voici quelques unes d'entre elles :

- Modification des classes kernel et boot

  La classe permettant de lancer la plateforme Madkit avait un constructeur privé dans la version originale ce qui nous empêchait de la lancer de l'extérieur. Nous avons rencontré le même problème dans le cas des fonctions pour lancer les agents de MadkitKernel qui n'étaient pas visibles depuis un package extérieur.

- Problème avec les wizards

  Lorsque l'on crée des wizards (partie développée dans la suite), ceux-ci se rajoutent dans le menu principal d'Eclipse (Exemple : New → Other → Madkit → Madkit Plugin).

  Ceux-ci sont donc visibles et « lançables » de n'importe quelle perspective. La gestion de la création d'un plugin nécessitait d'être en perspective Madkit sinon cela générait une exception, un objet ne se retrouvant pas initialisé. Nous avons donc mis en place un système de « switch » entre les perspectives (passer de la perspective initiale à celle de Madkit) lors de l'utilisation du wizard (de la même manière que ce qui est fait par Eclipse lorsque l'on crée un projet java par exemple).
  Ce changement de perspective est réalisé lors de l'initialisation du wizard :

```
public void init(IWorkbench workbench, IStructuredSelection selection)
{
	…
	try
	{
		workbench.showPerspective("org.eclipse.ui.madkitPerspective",
					workbench.getActiveWorkbenchWindow()) ;
		// Affichage de la perspective Madkit dans le workbench courant
	}
	catch (Exception ex)
	{
		//Erreur lors du changement de perspective
		…
	}
	…
}
```

- Evénements souris et « build »

Ensuite, deux problèmes nous paraissaient particulièrement urgents à corriger au niveau du Designer (permettant la gestion des plugins Madkit créés) :

- le fait que les évènements souris n'étaient plus du tout gérés.
- la construction (compilation) d'un projet créé via Madkit aboutissait sur un échec.

Nous avons passé deux jours sur ces problèmes. Nous nous sommes beaucoup documentés sur le passage de Swing à SWT.

En ce qui concerne la gestion des évènements souris, il est apparu que la solution était dans l'une des notes de la documentation des classes permettant cette transition. Une classe de Java permet le passage de AWT/Swing à SWT : **SWT_AWT** du package `org.eclipse.swt.awt` fourni par Eclipse. Le seul problème est que celle-ci ne gère pas les évènements en question si ceux-ci sont gérés directement par la fenêtre. Il faut donc rajouter un panneau dans lequel nous placerons tous les éléments que nous voulions mettre directement dans la fenêtre. La gestion sera alors effective.

Le problème qui faisait que la construction d'un projet créé via Madkit aboutissait à un échec venait du fait que le constructeur ne trouvait pas le compilateur java pointé par le *JAVA_HOME**. Malgré le fait de l'avoir défini dans le système comme d'environnement, cela ne fonctionnait toujours pas. Le problème a été résolu en changeant de JRE dans le Eclipse qui lançait le nouveau Eclipse où était attaché notre plugin. Il fallait prendre celui qui se situe dans le dossier JDK et dans un JRE seul.

Une fois ce problème résolu, la fonction build du designer avait un bug. Nous avions une erreur qui venait du fait que le Designer nous faisait changer de classloader pendant l'activation de la compilation et l'agent cessait de reconnaître son environnement et donc ne fonctionnait plus. Nous avons donc redéfini cette méthode dans la classe du GUI afin qu'elle devienne adaptée à l'exécution dans Eclipse. Elle rajoute le jar du projet compilé dans le classloader du plugin Eclipse qui est récupéré de l'objet (passé au constructeur) de notre environnement d'exécution d'Eclipse (méthode décrite plus loin).

Nous n'avons pour l'instant seulement fait une étude des deux plugins, choisi notre base de travail (partir du plugin de Christian Plutis), réalisés quelques ajustements mais aucune nouvelle fonctionnalité n'est effective. Nous allons donc découvrir la grosse partie de notre travail dans la suite de ce rapport.

1.2 – Apprentissage

Nous avons eu besoin, pour maintenant mieux s'approprier le projet et apporter notre touche personnelle au plugin Eclipse, de parfaire notre apprentissage sur des points importants.

Agent :

Comme on l'a dit précédemment, Madkit est une plateforme basée sur la notion d' « agent », il était important de bien connaître le but, le fonctionnement d'un agent. Pour cela, le module « programmation agent » nous était très utile.

Ant :

La modification des fichiers jar de Madkit, la compréhension de la construction des agents Madkit (build) a nécessité une initiation puis l'utilisation du logiciel ***ant**** (automatisation de projets, plus particulièrement Java) que nous ne connaissions pas auparavant.

Eclipse :

Comme pour Madkit, une connaissance plus approfondie d'Eclipse était bien entendu indispensable : création de ***plugins****, ***perspectives****, ***vues***, ***wizards*,*** gestion des projets sous Eclipse, …

Madkit :

Il a été essentiel que nous comprenions et étudions « en profondeur » le fonctionnement de la plateforme Madkit : la philosophie où tout est agent, ses classes principales, sa programmation, … En effet, comme nous le verrons par la suite, nous avons modifié, étendu des agents ce qui implique un minimum de bases sur ces derniers points.

Nous allons maintenant présenter les nouvelles fonctionnalités ou extensions du plugin Eclipse.

2 – Améliorations

2.1 – Les perspectives et les views

Au lancement d'Eclipse, une seule fenêtre s'ouvre contenant son ***workbench**** (environnement de travail). Celui-ci contient des ***perspectives****. Avec ce plugin, une sera dédiée pour travailler des projets Madkit.

Pour créer une nouvelle perspective, il faut tout d'abord ajouter une extension à notre fichier *plugin.xml* (Tout plugin Eclipse possède un fichier plugin.xml où sont décrites ses extensions. Le logiciel l'utilise à son lancement).

```
<extension point="org.eclipse.ui.perspectives">
     <perspective
          class="madkit.perspectives.MadKitPerspective"
          icon="icons/designeragent.gif"
          id="org.eclipse.ui.madkitPerspective"
          name="Madkit">
     </perspective>
</extension>
```

Les principaux attributs de cette extension sont les suivants :

id : un nom unique pour identifier la perspective
name : un nom utilisé dans les fenêtres permettant l'ouverture
de perspectives.
class : le nom complet de la classe définissant cette perspective.
icon : le nom complet de l'icône qui sera associée à cette perspective.

Il reste une étape à la création de notre perspective, définir une classe relative à cette perspective.

Cette classe doit implémenter l'interface *IPerspectiveFactory*. La seule méthode de cette interface est *createInitialLayout(IPageLayout layout)*.

Cette méthode va créer le layout initial, on y rajoute nos ***views**** en les plaçant dans le layout.

Ce placement se réalise selon trois types de disposition possibles (*bottom, left, right)* par rapport à un autre « objet » déjà présent.

Dans notre perspective Madkit, nous avons placé les vues par rapport à l'éditeur central.

```
String editorArea = layout.getEditorArea();
// on récupère l'éditeur central
```

On peut ajouter plusieurs vues à une même sous fenêtre, celles-ci apparaissant sous forme d'onglets. Pour cela, on crée un *IFolderLayout* qui fait office de « conteneur » de vues.

```
IFolderLayout topLeft = layout.createFolder("topLeft",
                        IPageLayout.LEFT,0.26f,editorArea);
```

Ici, on crée un folder à la gauche de l'éditeur central. 0.26f représente le ratio (entre 0 et 1) de placement. Dans ce cas, le folder se trouve en haut de la fenêtre.

A ce folder, on rajoute nos vues de la façon suivante :

```
topLeft.addView("org.eclipse.jdt.ui.PackageExplorer");
topLeft.addView("madkit.views.GroupObserverView");
```

Dans cet exemple, on insère deux vues : « ***Package Explorer**** » (vue déjà existante dans Eclipse) et « ***Group Observer*** » (vue que l'on a créée).

Dans notre cahier des charges, l'un des objectifs était d'intégrer dans Eclipse les principales vues de Madkit.

La perspective Madkit était déjà définie avec une vue « ***Designer**** ».

Mais cette vue n'était pas conforme à notre souhait, celui de faire utiliser, à nos vues, les agents Madkit (on conserve ainsi la philosophie de Madkit, basée sur l'agent). En effet, cette vue essayait de reproduire ses fonctionnalités mais n'avait aucun rapport avec l'agent Designer de Madkit. (propre arborescence, propres actions, autre design, …).

Nous avons donc créé une nouvelle vue Designer puis les suivantes : « ***Documentation****** », « ***Explorer**** » « ***Group Observer****** » et « ***Output**** ».

Il est important de noter toutefois que la vue Designer n'utilise pas l'agent standard développé dans Madkit mais un agent Designer spécialement adapté pour Eclipse. (Les explications sont développées dans une autre partie qui se trouve dans la suite de ce rapport). Cela pour laisser la plupart du travail à Eclipse et non à Madkit.

Nous allons nous intéresser dans cette partie aux étapes nécessaires à la création puis à l'ajout d'une nouvelle vue.

On déclare une nouvelle extension dans le fichier *plugin.xml*.

```
<extension point="org.eclipse.ui.views">
     <view
          name="Group Observer View"
          icon="icons/groupObserver.gif"
          class="madkit.views.GroupObserverView"
          id="madkit.views.GroupObserverView">
     </view>
     … autres vues
</extension>
```

Les principaux attributs de cette extension sont les mêmes que pour une perspective, leur effets portant sur la vue bien évidemment.

Nous allons voir comment cela se traduit dans l’implémentation des classes définissant les « views ».

La pratique courante est d’étendre la classe abstraite *ViewPart* pour hériter de sa principale fonctionnalité qui est de créer une nouvelle vue.

La méthode importante à redéfinir est *createPartControl(Composite parent)* présente dans l’interface *IWorkbenchPart*. Elle permet de changer le layout du Composite placé en paramètre « parent » et de lui donner des actions. C’est dans celle-ci que nous allons faire appel à l’agent caractéristique de la vue.

Il faut souligner que c’est le workbench qui s’occupe d’appeler cette méthode lorsqu’il en a besoin et non la classe cliente.

```
public void createPartControl(Composite parent)
{
     MadKitFacade.startJavaAgent("madkit.system.GroupObserve
                                  r",parent);
     // On démarre l’agent Group Observer
}
```

MadkitFacade est une classe qui s’occupe de réaliser les principales actions de Madkit (créer un plugin, créer un agent, lancer un agent, …).

Nous avons rencontré un problème dans le domaine des perspectives. En effet, comme nous le verrons plus tard, nous avons eu le besoin qu'Eclipse « reconnaisse » les plugins Madkit comme des projets Java. Notre but était donc d'essayer d'avoir les ressources nécessaires à la gestion de projets Java. Notre première idée a donc été de permettre à notre perspective Madkit d'hériter de la perspective Java. Nos recherches nous ont montrées qu'un héritage comme une classe (du style *extends*) n'était, à priori, pas possible. Par contre, la solution serait l'extension d'une perspective (ici Java), c'est-à-dire lui rajouter des vues.

Voici comment cela se caractérise dans le fichier *plugin.xml*.

```
<extension point="org.eclipse.ui.perspectiveExtensions">
     <perspectiveExtension
     targetID="org.eclipse.jdt.ui.JavaPerspective">
          <view
               id="madkit.views.GroupObserverView"
               relative="org.eclipse.jdt.ui.PackageExplorer"
               relationship="stack" />
     </perspectiveExtension>
</extension>
```

targetId : représente l'id de la perspective que l'on va étendre. (Java dans notre cas)

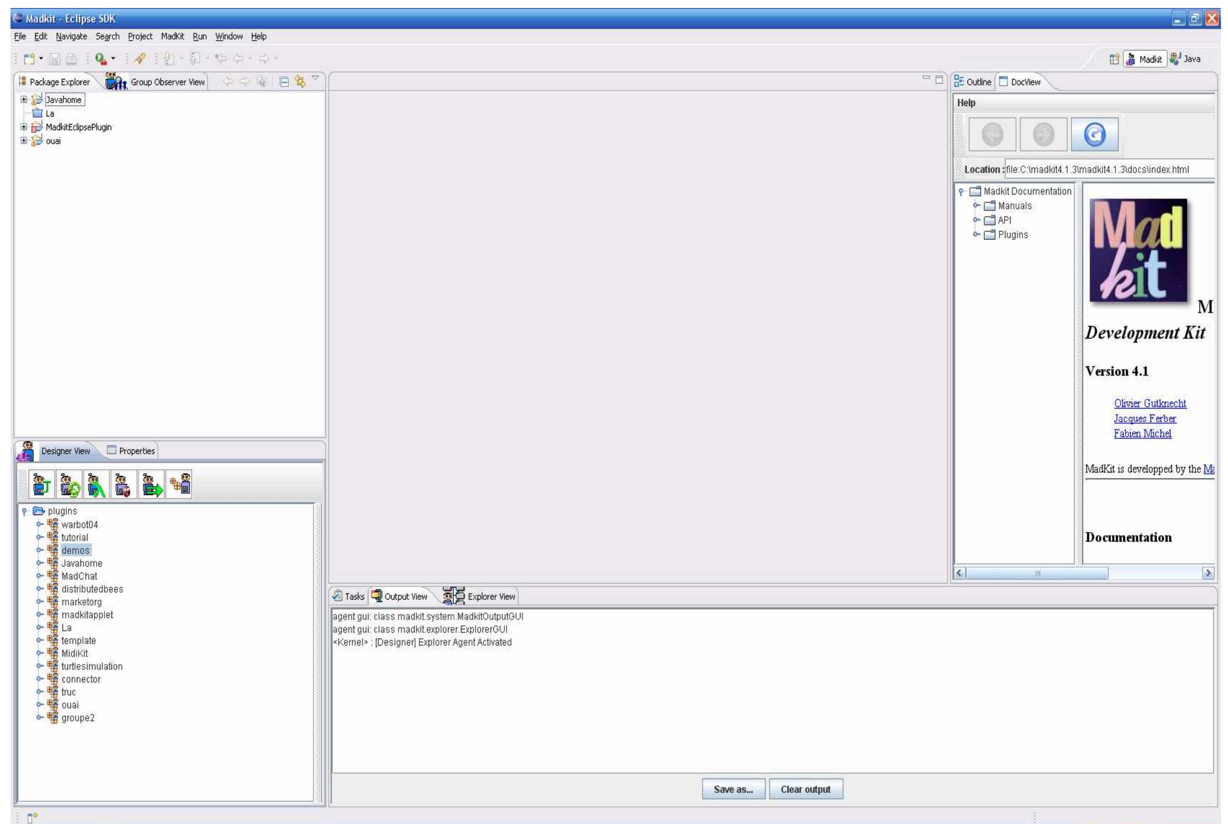
Ensuite, on déclare les vues que l'on souhaite rajouter (dans cet exemple, Group Observer).

relative : on indique la vue à partir de laquelle on va placer la nouvelle.

relationship : concerne le placement (ici, stack veut dire placement à la suite de la vue « relative » dans le même folder).

Le défaut de cette solution était qu'à ce moment là, la vue java possédait les vues Madkit (plus de perspective Madkit) même lorsque l'utilisateur ne souhaitait seulement développer un projet java et non Madkit.

Finalement, en remettant la perspective Madkit (avec la vue Package Explorer en plus), nous sommes arrivés à ce que nous désirions sans avoir à étendre celle de Java.



Vue de la perspective Madkit

2.2 – Création de projets java

La principale utilité de ce plugin est bien entendu de pouvoir éditer les fichiers dans Eclipse et donc d'utiliser tous ses avantages (signalement des erreurs, complétion automatique avec le raccourci *Control+Espace*, nombreux autres raccourcis claviers, coloration syntaxique, …). Mais pour cela, il faut qu'Eclipse considère le fichier comme appartenant à un projet Java.

Notre but a donc été de créer un projet java à partir des sources d'un plugin Madkit et non dans le ***workspace****. Comme le permet Eclipse avec son wizard lorsqu'on développe un projet Java et que l'on choisit l'option *« from existing sources »*. Il faut donc maintenant le réaliser par un programme.

La première étape est de créer un projet Eclipse (*IProject)*

```
IProject project =
      ResourcesPlugin.getWorkspace().getRoot().getProject(
            pluginName);

// On va fournir une description
IProjectDescription description =
     project.getWorkspace().newProjectDescription(
            project.getName());
```

Le problème, à cet instant, est que le projet n'est pas encore localisé à l'endroit désiré mais dans le workspace.

```
description.setLocation(projectPath);
project.create(description,progressMonitor);
project.open(progressMonitor);
```

On a indique un emplacement différent à la description du projet (projectPath : chemin du plugin). Ensuite, on peut le créer puis l'ouvrir.

La seconde étape consiste à ce que ce projet devienne de type Java (***Java nature*)***.

Pour cela, une partie du code a pour travail de changer la nature (*JavaCore.Nature_ID* représentant la Java Nature).

```
String[] prevNatures = description.getNatureIds();
String[] newNatures = new String[prevNatures.length + 1];
System.arraycopy(prevNatures, 0, newNatures, 0,
     prevNatures.length);
newNatures[prevNatures.length]= JavaCore.NATURE_ID;
description.setNatureIds(newNatures);
```

On peut maintenant finaliser le typage de notre projet en Java.

```
IJavaProject javaProject;
javaProject = JavaCore.create(project);
```

Dans la dernière, il nous faut ajouter les librairies nécessaires au bon fonctionnement du projet. La principale étant celle du ***JRE****. Dans notre cas, nous ajoutons deux librairies de Madkit nécessaires au début d'un nouveau plugin. On va donc modifier le ***classpath**** du projet.

```
IPath jreLibPath = new Path(javaHome.getPath()).
                  append("lib").append("rt.jar");
// Librairie du JRE

String libDir = MadKitData.getMadKitRoot() +
     File.separator+MadKitData.MADKIT_LIB_DIR;
// Librairies de Madkit

String [] tabVar = {"JRE_LIB","Madkit_LIB"};
IPath [] tabPath = {jreLibPath,new Path(libDir)};
JavaCore.setClasspathVariables(tabVar,tabPath,null);
// Ajout des variables au classpath

javaProject.setRawClasspath(new IClasspathEntry[]
{
     JavaCore.newSourceEntry(srcFolder.getFullPath()),
     JavaCore.newVariableEntry(
               new Path("JRE_LIB"),
               new Path("JRE_SRC"),
               new Path("JRE_SRCROOT")
     ),
     JavaCore.newLibraryEntry(
          new Path(libDir +
               File.separator+"madkitkernel.jar"),null,null),
     JavaCore.newLibraryEntry(
          new Path(libDir + File.separator +
               "madkitutils.jar"), null, null)
},null);
//modification des entrées du classpath
```

Il faut souligner qu'Eclipse nous génère deux fichiers xml : .project et .classpath.

2.3 – Communication inverse

Description du besoin :

On avait au début un plugin qui avait pour fonctionnalité de démarrer le noyau de Madkit et de lancer des agents dans son environnement. Au cours de l'avancement, on a eu besoin de faire la communication inverse : on voulait invoquer une fenêtre wizard Eclipse, quand on cliquait sur un bouton correspondant, au lieu d'avoir la fenêtre de base de Madkit et, au double clic sur un fichier source, celui-ci devait s'ouvrir dans la fenêtre d'édition d'Eclipse.

Première approche :

Deux solutions ont été essayées au début sans succès :

- Modifier le classloader du plugin en lui rajoutant tout le dossier d'Eclipse. Cela a pour effet de nous permettre d'utiliser les bibliothèques propres à Eclipse pendant l'exécution de l'agent, la compilation par contre doit se faire sous Eclipse en rajoutant dans le classpath les bibliothèques utilisées.
  Ainsi, on pourrait créer le wizard dans le code de l'agent. Le problème est que dans SWT, un seul thread peut avoir accès aux ressources UI, donc une invocation n'est pas possible depuis un autre thread.

- Utiliser la synchronisation wait/notify : c'est-à-dire attendre un objet synchronisé que l'on passe à l'agent, quand une action se produit l'agent devrait faire notify() sur cet objet. Cette solution n'a pas fonctionné car Java interdit de notifier ou attendre un objet qui n'appartient pas au thread.

Solution : ActionInterpreter

La solution que nous avons trouvée consiste à lancer l'agent différemment avec un constructeur qui prend un objet de notre environnement d'exécution d'Eclipse en paramètre. Cet objet est une instance de la classe *ActionInterpreter* qui possède une méthode *invokeAction* qui prend une ou deux chaînes de caractères en paramètre, selon les actions, et qui lance l'exécution du code correspondant à l'intérieur du thread Eclipse qui s'occupe de l'UI (méthode *Display.asynchExec(Runnable)*).

Du coté de l'agent, pour une meilleure réutilisabilité, on implémente la méthode *sendEclipseRequest* qui, par réflexivité, va récupérer la classe et le classloader de l'objet Eclipse. Ensuite, avec ce classloader, on charge la classe appropriée et on appelle la méthode statique *invokeAction* sur cette classe avec les paramètres variant selon l'action que l'on veut exécuter. Donc, en redéfinissant *actionPerformed* dans notre GUI (héritant du GUI original) ou un noeud correspondant au fichier source, on appelle *sendEclipseRequest* de l'agent propriétaire.

2.4 – Modification du Designer

Il nous a été demandé de modifier le Designer dans Eclipse de façon à ce que seule l'arborescence soit visible au départ et que les propriétés n'apparaissent que lorsque l'utilisateur le demande.

Il a d'abord fallu modifier le code de création des composants visuels en deux cas différents : si l'on est sur Madkit, on veut utiliser l'interface graphique du Designer normal, sinon (on est donc sur Eclipse), on se servira de celle adaptée à Eclipse :

```
public void initGUI()
{
     if(System.getProperty("eclipse.startTime") == null)
          gui = new PluginDesignerGUI(this);
     else
          gui = new EclipsePluginDesignerGUI(this);
     setGUIObject(gui);
}
```

Celle-ci, comme vu ci-dessus, s'appelle « EclipsePluginDesignerGUI », elle héritera bien sûr de PluginDesignerGUI afin d'en garder le comportement global, mais maintenant n'affichera plus le panneau des propriétés, il faudra maintenant faire clic droit sur un projet et choisir « Properties ».

Le panneau des propriétés est utilisé à trois endroits dans la classe PluginDesignerGUI :

- A la création : la vue « propriétés » est affichée mais les valeurs ne sont par remplies
- Lors de la sélection de l'un des plugins dans l'arbre
- Lors du clic droit et du choix de « Properties »

Il a donc fallu, dans le nouveau constructeur, bloquer l'affichage du panneau, ceci grâce à la méthode *setVisible(boolean)* qui permet de ne pas dessiner le composant l'invoquant.

Pour les deux autres utilisations de la vue « propriétés », on se rend compte qu'elles font toutes les deux appel à la méthode *showPlugin(String name)*, nous avons donc dû surcharger celle-ci pour que, une fois les informations sur le plugin à afficher récupérées, elle montre la vue « propriétés » dans une nouvelle fenêtre.

Puis afin de modifier le comportement, nous avons également redéfini les fonctions *build* et *actionPerformed*. Pour pouvoir ouvrir un fichier source dans Eclipse, nous avons fait hériter des classes de *PluginTree* et de *EditNode* en y redéfinissant les actions à prendre.

2.5 – Modification des actions du plugin

Une fois qu'on a pu établir la communication entre Eclipse et l'environnement Madkit, il nous fallait adapter des nouvelles fonctionnalités d'Eclipse au Designer.

2.5.1 – Les boutons du designer :

Les boutons du Designer permettent la création de nouveaux plugins ou agents de plusieurs types (java, scheme, …). Le Designer possède déjà ces fenêtres pour assister à la création mais il était souhaitable pour ce plugin que l'agent lance des wizards Eclipse.

Il nous était demandé d'utiliser des ***wizards*** le plus générique possible dans le souci de faciliter une extension future.

Nous allons tout d'abord détailler le mode de fonctionnement pour insérer un wizard Eclipse dans un plugin puis nous nous attarderons plus précisément sur notre application (pour les plugins puis pour les agents).

On utilise la même démarche que pour les perspectives et les views, c'est-à-dire, l'ajout d'une extension dans *plugin.xml*.

```
<extension point="org.eclipse.ui.newWizards">
     <wizard
          class="madkit.wizards.PluginWizard"
          hasPages="true"
          icon="icons/plugin.gif"
          id="madkit.wizards.PluginWizard"
          name="MadKit Plugin Wizard"/>
</extension>
```

L'attribut nouveau est *hasPages* où l'on indique si le wizard possède au moins une page. On va voir par la suite ce qu'est exactement une page.

On code ensuite la classe qui définit le wizard. Celle-ci hérite de la classe abstraite *Wizard* et implémente l'interface *INewWizard*.

Nous redéfinissons deux méthodes très utiles de la classe *Wizard* qui sont

- *addPages() :* c'est ici que l'on ajoute nos pages à notre wizard.

```
public void addPages()
{
     page = new AgentWizardPage(selection,target);
     //déclaration d'une page
     addPage(page);
     // ajout de la page
}
```

- *performFinish() :* c'est ici que l'on gère les actions à effectuer lorsque l'utilisateur a fini de naviguer à travers les pages du wizard. (en résumé quand il appuie sur le bouton de fin).

```
public boolean performFinish()
{
     try
     {
                    …
          MadKitFacade.createAgent(agentName, antTarget,
               pluginName, agentType, extension);
          // creation d'un agent
     }
     …
}
```

- **<u>Création de plugins :</u>**

L'action NewPluginAction provoque l'ouverture d'un wizard (voir partie apprentissage) de création d'un nouveau plugin madkit.

Il a fallu lancer cette action dans la méthode *run()* de la classe *ActionTranscript*, mais cela ne suffisait pas car, après sa création, le nouveau projet n'était pas présent dans l'arbre du Designer.

Pour réaliser la mise à jour, à l'aide de la réflexivité on lance *PluginAgent* qui se charge de signaler au Designer qu'il y a un nouveau projet. Voici le code :

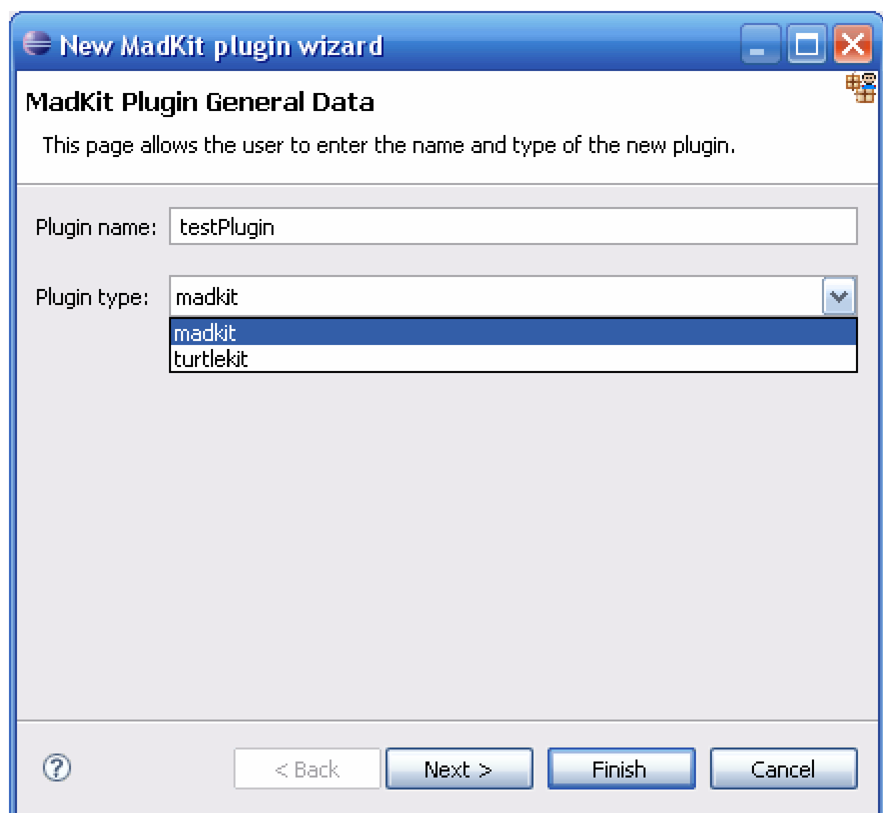
```
Class<?> agentClass =
     EclipseMadkitClassLoader.getInstance().
     loadClass("madkit.system.PluginAgent");
Constructor<?> cons = agentClass.getConstructor(
     new Class[]{});
Object agent = cons.newInstance(new Object[0]);
Method initMeth = agentClass.getMethod("init", new
     Class[]{File.class});

File newFile = new File(new File(
     MadKitData.getMadKitRoot(),
     MadKitData.MADKIT_PLUGIN_DIR),pluginName);

initMeth.invoke(agent, newFile);

launchAgent(agent);
```

Et pour finir il faut modifier le projet pour qu'il soit reconnu comme projet java par Eclipse (cette opération est décrite plus haut).



Wizard de création d'un plugin

- **Création d'agents :**

Il existait déjà un wizard pour la création d'un agent. Le problème était que ce wizard n'était pas tout à fait compatible avec la modification du Designer.

En effet, la détection du projet sélectionné (projet courant) était adaptée à l'ancien Designer personnalisé.

Nous avons donc utilisé le système de communication mis en place entre le Designer et Eclipse (développé dans une autre partie).

Le plugin en cours qui est actualisé dans la méthode *showPlugin(String name)* de PluginDesignerGUI.

```
void showPlugin(String name)
{
     PluginInformation
          info=(PluginInformation)plugins.get(name);
     currentPlugin = info;
     …
}
```

Dans le *actionPerformed()*, lorsque l'action correspond à la demande d'un nouvel agent, on récupère le plugin sélectionné puis on envoie l'information à Eclipse. Si aucun plugin n'est sélectionné, l'utilisateur est informé de devoir en choisir un au préalable.

```
target = currentPlugin.getName() ;

ag.sendEclipseRequest(new
     Class[]{String.class,String.class},new
          Object[]{c,target});
// c correspond à la commande, target au plugin en cours
```

Une fois que nous avons récupéré le nom du plugin dans Eclipse, nous pouvons le transmettre au wizard, qui le transmet lui aussi à son unique page. Le plugin cible dans lequel l'agent sera créé pourra être modifié via un bouton « Browse » qui propose l'arborescence.

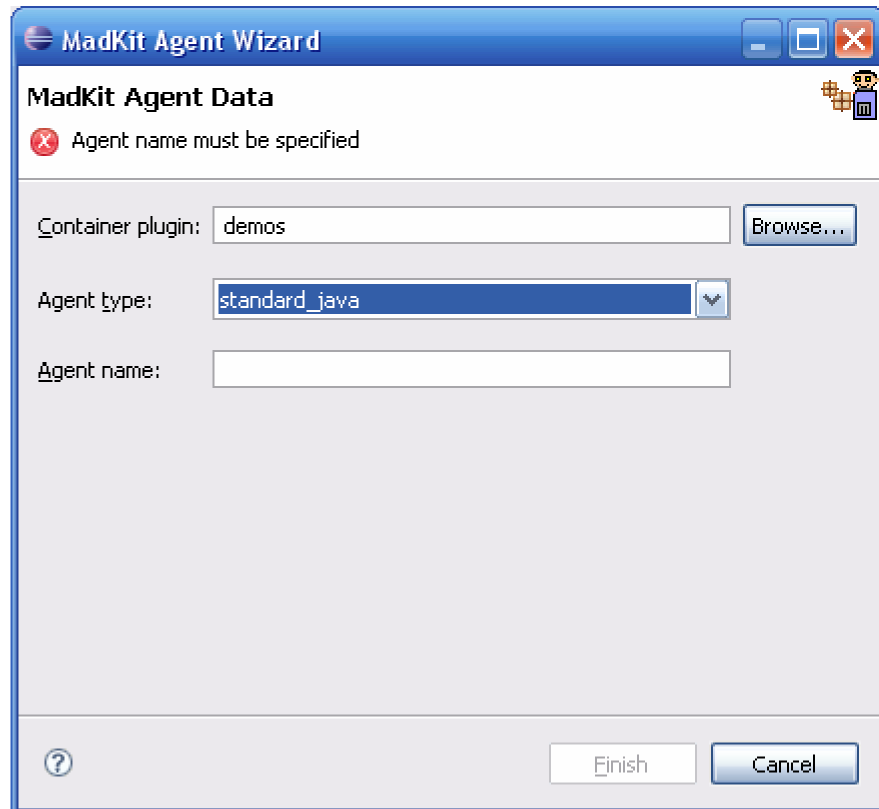
Des erreurs empêchent la validation par l'utilisateur si il n'y a aucun nom d'agent indiqué, si celui existe déjà ou si le plugin cible n'est pas un projet Madkit.

Une fois les informations sur l'agent saisies, l'action principale de *performFinish()* consiste à demander à Madkit de créer cet agent.



pop-up si aucun plugin n'est sélectionné

ou



Wizard de création d'un agent

2.5.2 – Edition des fichiers source dans l'éditeur d'Eclipse

Pour que les fichiers sources (.java) s'ouvrent dans Eclipse, il fallait utiliser l'action correspondante en y rajoutant un paramètre (le nom du fichier) en constructeur.

Cela avait pour effet de faire ce que l'on souhaitait à part que nous ne pouvions pas utiliser les fonctionnalités intéressantes d'Eclipse car le fichier était considéré comme ne faisant pas partie d'un projet.

Donc on récupère le dossier racine du projet, vérifie si il n'est pas considéré comme projet java et on crée un nouveau projet dans ce dossier (on a vu auparavant sa méthode de construction).

3 – *Bilan critique*

3.1 – Organisation

Pour faciliter le suivi des mises à jour dès que nous avions une version fonctionnelle, nous nous sommes servi d'un serveur *svn*. Ce dispositif permet de facilement récupérer ou envoyer des mises à jour. Il contrôle aussi les conflits entre les versions : lorsque l'on essaye de mettre à jour un fichier sans avoir sa dernière version ou si au contraire, on tente de mettre à jour un fichier que l'on a modifié sans l'avoir précédemment mis à jour.

Le chef de projet s'occupait de la communication en général : répartition du travail, formation des groupes, communication avec l'encadrant et la personne qui a réalisé le plugin que l'on a pris comme base.

Pour ce qu'il s'agit des groupes, ils changeaient assez souvent selon ce que chacun souhaitait faire. La plupart de temps, nous travaillions en binôme. Dans le cas où nous rencontrions plus de difficultés que prévu, nous nous y mettions à plus de personnes.

3.2 – Tests réalisés

Pour tester notre plugin, le moyen le plus efficace était de régulièrement l'utiliser lors de nos TPs du module de programmation agent.

Ceci nous mettait en situation de simple utilisateur Madkit. Nous n'étions plus dans « l'esprit » concepteur qui peut orienter nos tests dans une certaine direction et nous faire passer à côté d'erreurs générées par des actions courantes.

Cela nous permettait également de faire chaque semaine une constatation effective de son évolution, de ses améliorations à apporter.

Nous essayions lors de nos rencontres hebdomadaires avec nos encadrants de leur amener une démonstration de notre travail, nous avons ainsi fait ressortir des erreurs que nous ne soupçonnions pas avant un test par un utilisateur non concerné directement par sa programmation.

Pendant la finalisation du projet, une version a été envoyée à nos tuteurs pour la tester sur leur machine.

3.3 – Objectifs atteints

Nous avions deux principaux objectifs au début de ce TER, qui étaient d'intégrer les vues principales de Madkit dans Eclipse puis de pouvoir éditer des fichiers appartenant au plugins dans l'éditeur Eclipse en profitant de toutes les possibilités qu'il nous offre.

On peut dire que ces objectifs sont pratiquement entièrement satisfaits. En espérant que cela incite maintenant à utiliser Madkit sous sa nouvelle forme de plugin Eclipse.

3.4 – Améliorations possibles

Par manque de temps, certaines opérations n'ont pas été effectuées et font partis d'améliorations envisageables :

Une première et assez utile serait la gestion de l'ouverture et de l'édition dans Eclipse des fichiers lancés à partir de la vue Explorer. (comme cela est possible sur le Designer)

Nous penserions également à intégrer le contenu de la fenêtre propriétés dans le wizard.

Une autre serait de sortir les boutons du designer et les placer dans la barre d'outils d'Eclipse (plus de place disponible) permettant une extension des possibilités plus importante.

Bien évidemment, nous ne les avons pas toutes imaginées, d'autres pourraient sûrement être proposées.

Glossaire

- ***Ant :*** projet Apache principalement utilisé pour automatiser la construction de projets (principalement en langage Java pour ant) à l'instar des logiciels Make.
- ***Classpath :*** paramètre passé à une machine virtuelle Java qui définit le chemin d'accès au répertoire où se trouvent les classes et packages afin qu'elle les utilise.
- ***Eclipse :*** environnement de développement universel, notamment le plus utilisé en ce qui concerne le langage JAVA. Il est principalement écrit en JAVA (à l'aide de la bibliothèque graphique ***SWT****).
  Son gros avantage est que son architecture est basée sur la notion de plugins : toutes ses fonctionnalités sont développées en tant que plugins, ce qui permet une indépendance entre l'application et le plugin.
- ***Java Builder :*** compilateur Java utilisé par Eclipse.
- ***JAVA_HOME :*** variable d'environnement indiquant le chemin du répertoire du ***JDK*** sur la machine.
- ***Java Nature :*** lorsqu'on l'ajoute à un projet, le classpath et le Java Builder sont configurés pour travailler sur ce projet.
- ***JDK (Java Development Toolkit) :*** environnement de compilation du code Java.
- ***JRE (Java Runtime Environment) :*** environnement d'exécution JAVA.
- ***MadKit :*** plate-forme multi-agents modulaire développée au LIRMM (par Olivier Gutknecht, Jacques Ferber et Fabien Michel). Elle est écrite en JAVA et est basée sur le modèle AGR (Agent/Groupe/Rôle), c'est-à-dire des agents appartiennent à des groupes et jouent des rôles.
- ***Package Explorer :*** Dans cette vue, on trouve la liste des projets référencés dans le Workspace.
- ***Perpective :*** Une perspective présente une partie du projet de développement dans un certain angle de vue. Elle est composé de sous-fenêtres de deux types différents :
  - vues spéciales (***views***)
  - éditeurs

  Ces « outils » disponibles répondent bien sûr aux besoins du projet en cours. (la perspective Java permet d'écrire du code java, la perspective Debug possède des vues permettant le débogage de programmes java, …)
- ***Plugin :*** programme qui interagit avec un logiciel principal, appelé *programme hôte* (ici Eclipse) pour lui apporter de nouvelles fonctionnalités. Ceci est également le nom des projets sous Madkit.

- *Programmation orientée agent : programmation* dans des systèmes multi-agents.
  Un système multi-agent étant un ensemble d'entités (appelées *agents*) autonomes situées dans un environnement et interagissant selon une certaine organisation. Ces agents peuvent être des êtres humains, des robots, des processus, ...
- *SWING :* bibliothèque graphique pour Java basée sur le modèle MVC (Model-View-Controller).
- *SWT (Standard Widget Toolkit) :* bibliothèque graphique pour Java.
  N'implémente en Java que les fonctionnalités qui ne sont pas offertes par les toolkits sous-jacents, économise donc les ressources, d'où sa rapidité d'exécution par rapport à *Swing**. *Eclipse** repose sur cette architecture.
- *View :* Une vue permet de visualiser, faire interagir les informations au sein du workbench (la vue « Properties » va afficher les propriétés du projet ou du fichier sélectionné, d'autres vont permettre de naviguer au sein d'un projet, …).
  Ces vues peuvent posséder leurs propres menus, barre d'outils, boutons.
- *Vue Designer :* permet de voir tout les plugins existants et leurs composants sous la forme d'une arborescence. Elle permet également de créer un nouveau plugin ainsi qu'un nouvel agent dans ce plugin.
- *Vue Documentation :* permet d'accéder à la documentation MadKit (les différents manuels, l'API et une présentation des différents plugins implémentés dans MadKit).
- *Vue Explorer :* affiche la structure et le contenu des différents répertoire de Madkit. Permet de nombreuses possibilités sur ces derniers.
- *Vue Group Observer :* permet d'observer l'organisation générale (c'est-à-dire les communautés, groupes et rôles) et les évènements qui se déroulent dans MadKit (par exemple, la création de groupes, les agents qui entrent ou quittent un groupe).
- *Vue Output :* affiche le texte « imprimé » par les commande telles que System.out.println(), … . Equivalent de la console sous Eclipse.
- *Wizard :* assistant (pour la création de projets, classes, …)
- *Workbench :* le workbench Eclipse désigne l'environnement de travail dans lequel l'utilisateur va travailler. Il est composé de menus, barres d'outils, et d'une ou plusieurs *perspectives*. Toutefois, seulement une ne peut être affichée à la fois.
- *Workspace Eclipse :* espace de travail d'Eclipse comme son nom l'indique, endroit où se trouvent les projets et les fichiers manipulés par le logiciel.
  Par ailleurs, on peut tout de même créer des projets à partir de sources existantes dans n'importe quel répertoire.

Références

Voici quelques références qui nous ont été utiles pour réaliser ce plugin :

- Cours de programmation agent
- Cours de méta-programmation récursive – réflexivité en java
- http://www.eclipse.org – informations générales et quelques tutoriels
- http://www.developpez.com – bonne documentation SWT
- http://www.eclipsetotale.com – plugins
- http://java.sun.com – API Java
- http://www.docjar.com – pour les sources
- http://mep.freehostia.com. – site dédié au plugin de Christian Plutis
- http://madplugin.neuf.fr/ – site dédié au plugin de Quentin Caron

Remerciements

Nous souhaitons dans cette partie adresser nos remerciements :

Tout d'abord à nos tuteurs de projets M. Jacques Ferber et M. Tibériu Stratulat qui ont suivi notre travail, nous ont aidés, indiqués des pistes à approfondir. Ils nous ont également donné quelques bases d'ingénierie logicielle spécifique à Java.

Ensuite, à M. Christian Plutis, auteur du plugin sur la base duquel nous avons réalisé notre version, qui nous a toujours apporté de l'aide en cas de besoin (communication par messagerie).

Ils sont également adressés à M.Quentin Caron pour nous avoir fourni la version de son plugin.

Pour finir, à M. Michel Leclère pour le suivi qu'il donne aux TER : répartition, planning, recueil des cahiers des charges, des rapports, soutenances, …

Annexe

Manuel d’utilisation du plugin pour Eclipse permettant l’utilisation de l’environnement Madkit :

- **Lancement :**

Pour que le plugin fonctionne, il faut simplement copier le fichier .jar fournit par nos soins dans le dossier plugins de Eclipse. Ensuite, il suffit de lancer l’environnement de programmation. Ceci effectué, il faudra changer de perspective (bouton en haut à droite) et utiliser celle nommée « Madkit » pour avoir toutes les vues implémentées pour ce plugin. Au premier lancement, une fenêtre s’ouvrira demandant le chemin du logiciel Madkit installé sur l’ordinateur. Il faudra donc l’avoir installé au préalable. Il est disponible à cette adresse http://www.madkit.org/ .

Il est aussi possible d’ouvrir les fenêtres des fonctionnalités indépendamment, en cliquant sur Window (dans le menu en haut) puis sur « Show view » → « Other ... » et enfin dans la fenêtre qui vient de s’ouvrir, il faut double-cliquer sur Madkit (ou sur le + à coté) et ensuite sélectionner la ou les vues voulues.

- **L’utilisation :**

Une fois lancé, il sera possible d’utiliser certaines fonctionnalités de l’environnement Madkit.

  - **Designer :**

La plus importante de celles-ci est le Designer. Elle se situe dans la partie gauche, en bas. C’est à partir de cette vue que l’on pourra créer des plugins pour Madkit, et des agents de divers types pour ceux-ci. L’arborescence présentée est celle des projets déjà présents dans le dossier plugins du logiciel.

Le premier bouton permet alors de créer ces fameux plugins. En cliquant dessus, un Wizard apparaîtra dans lequel il est demandé d’entrer un nom et de choisir le type du projet (pour le moment, il n’y a que deux choix : TurtleKit ou Madkit). Une fois ceci fait, il reste juste à cliquer sur le bouton « Finish ce qui engendrera la création et une première compilation du plugin. Notons, que ceci ajoute un dossier dans l’arborescence de la vue Designer et en crée un dans la vue Package Explorer. Ceci est nécessaire pour que les raccourcis d’Eclipse (Ctrl+Espace particulièrement) fonctionnent. Un agent par défaut est créé ayant le nom « nomduProjetHello.java » qui envoie « Hello World! » dans une console.

La création d'agents est maintenant possible pour le projet. Celle-ci se fait avec les autres boutons de la vue Designer (chacun représentant un type : Java, Python, Scheme, BeanShell, Jess). Pour ce manuel, nous allons nous intéresser à la création d'un agent Java (les autres types peuvent être construits de la même façon). Après avoir cliqué sur le bouton correspondant à cette dernière action, un assistant va s'ouvrir demandant d'entrer le nom et le type de ce dernier. Une combo box permet le choix entre diverses caractéristiques qui vont générer des codes différents (Ceux finissant par GUI seront pourvu d'une interface graphique en plus de la console). Une fois l'agent créé, en cliquant sur « Create », il sera placé dans le dossier src/madkit/nomduPlugin où il pourra alors être modifié dans Eclipse en double-cliquant dessus dans la vue Designer ou dans Package Explorer. La sauvegarde des modifications se fait simplement en cliquant sur « File » et « Save » de l'environnement de programmation.

Il sera alors possible d'utiliser les aspects pratiques d'Eclipse (Raccourcis et aide à la programmation) pour des agents Java créés directement sous Eclipse avec ce plugin ou alors par le logiciel Madkit.

- **Output View :**

Cette vue permet d'avoir le compte rendu des actions effectuées par les fonctionnalités de Madkit sous Eclipse. Pour ceux qui utilisent couramment Eclipse, on peut dire que cette vue est un équivalent de la vue Console. Elle se situe dans le bandeau en bas. Activée, elle permettra de voir les détails et le succès des actions effectuées ou les problèmes de compilation et les exceptions levées. C'est une aide à la programmation et surtout pour comprendre les erreurs effectuées.

Il est possible de sauvegarder tout le texte qui y est marqué en cliquant sur le bouton « Save as… » qui créera un fichier texte (.txt) à partir du contenu de celle-ci, permettant de garder une trace de tout ce qui s'est passé car cette fenêtre se remet à zéro quand on quitte et redémarre Eclipse .

Il est aussi possible d'effacer tous les messages qu'il y a dans la vue en pressant sur le bouton « Clear output » si celle-ci est trop encombrée.

- **Explorer View :**

C'est, comme son nom l'indique, l'explorateur pour Madkit. Il permet de voir facilement tout ce qui se situe dans le dossier de Madkit. Il se situe en bas à coté de la vue Output View.

On peut y faire des recherches en utilisant la commande « Find » du menu « File » et en entrant le fichier ou dossier recherché. Le nom, la taille, la date de dernière modification, si c'est un dossier ou pas et le chemin absolu seront affichés sous forme de liste.

Le menu « File » permet aussi de créer de nouveaux dossiers avec « Create Folder » et d'en effacer avec « Delete Folder ».

Le menu « Edit » de copier/couper et coller des fichiers et des dossiers d'un endroit à un autre dans le dossier de Madkit en utilisant les commandes « copy », « cut » et « paste ».

La commande « Detail » du menu « Display » permet d'afficher les détails de tout ce que contient le dossier sélectionné.

On peut lancer des agents directement par cette vue. Par contre, on ne peut pas modifier de fichiers .java directement de cette vue sous Eclipse ; il faut passer par le Designer ou le Package Explorer.

- ### Group Observer View :

Cette vue se situe dans la partie gauche, au dessus du Designer, à coté de l'onglet Package Explorer. Elle permet d'observer l'organisation générale (c'est-à-dire les communautés, groupes et rôles) et les évènements qui se déroulent dans MadKit (par exemple, la création de groupes, les agents qui entrent ou quittent un groupe).

Cette vue est divisée en deux parties :

- Celle du haut permet de voir tous les membres de tous les groupes crées dans les différentes communautés. La partie « public » est celle qui représente les agents utilisés par Madkit.

- Celle du bas permet de voir les différents messages échangés entre chaque agent. Il faut réaliser le fait que chaque fonctionnalité de Madkit est un agent, donc elle échange aussi des messages. On peut voir sous l'onglet « Actions » les évènements qu'elles ont générés.

- ### DocView :

Cette vue, située à droite de l'environnement Eclipse permet d'avoir une documentation complète sur Madkit. Il y a le détail de toutes ses fonctionnalités ainsi qu'une API complète sur toutes les librairies nécessaires pour la programmation orientée agent en utilisant Madkit.

Elle est présentée sous forme d'une page HTML découpée en deux parties : Celle de gauche avec les grands chapitres de la documentation et à droite avec le corps du document.

Manuel pour le développeur voulant modifier le plugin MadKit pour Eclipse

Un nouveau type d'agent MadKit a été mis en place pour pouvoir interagir dans l'environnement Eclipse, c'est-à-dire contribuer à son aspect visuel.

Pour mettre en place un tel agent il faut respecter les spécifications suivantes :

1. Côté Madkit

Cet agent doit posséder :

- une variable eclipseObject de type Objet.
- Un constructeur qui prend un Object en paramètres et initialise eclipseObject avec ce paramètre.
- La méthode suivante :

```
public void sendEclipseRequest(Class<?>[] types,Object[] objects)
{
	int lineIndex = 0;
	try
	{
		String clsName;
		if (eclipseObject == null)
		{
			System.out.println("Null argument received. Activation aborted.");
			return;
		}
		lineIndex++;
		clsName = eclipseObject.getClass().getName();
		ClassLoader ecl = eclipseObject.getClass().getClassLoader();
		Class agentClass = ecl.loadClass(clsName);
		lineIndex++;
		Method testMeth = agentClass.getMethod("dispatchAction", types);
		lineIndex++;
		testMeth.invoke(eclipseObject, objects);
	}
	catch (Exception ex)
	{
		System.out.println(">> reflection error: " + lineIndex);
		ex.printStackTrace();
	}
}
```

Note : l'agent PluginDesignerAgent est le seul qui implémente cette spécification au moment de l'écriture.

Exemple :

```
class TestEclipseAgent extends Agent
{
      protected Object eclipseObject ;

      TestEclipseAgent(Object o)
      {
            eclipseObject=o ;
            wait(3000) ;
            sendTest() ;
      }


      public void sendEclipseRequest(...) //voir la méthode plus haut

      //action de test avec un seul paramètre
      public void sendTest()
      {
            sendEclipseRequest(new Class[]{String.class},new
            Object[]{"test"});
      }
}
```

2. Côté Eclipse :

1. Lancer cet agent avec la méthode startLoopbackAgent(String name) de base.MadkitFacade où name est le nom complet de l'agent (incluant le package).
2. Rajouter du code dans le constructeur de la classe imbriquée ActionTranscript de actions.ActionInterpreter par rapport aux arguments du constructeur. Ces arguments correspondent à ceux qui vont être passés à sendEclipseRequest.
3. Redéfinir run() de ActionTranscript

Exemple :

```
//quelque part
MadkitFacade.startLoopbackAgent("madkit.TestEclipseAgent");

//dans ActionInterpreter
.........
class ActionTranscript implements Runnable
{
      ............
      public void run()
      {
            ...........
            else if(action.equalsIgnoreCase("test")
            {
                  //notre action: ouvre une popup
                  MessageDialog.openInformation(Display.getDefault().
                  getActiveShell(), "Test");
            }
      }
}
```